Ouch, Those Programs Are Painful!

by Don Libes

Expect is a program to control interactive applications. These applications interactively prompt and expect a user to enter keystrokes in response. By using Expect, you can write simple scripts to automate these interactions. And using automated interactive programs, you will be able to solve problems that you never would have even considered before. In this article Don Libes, the author of Expect, gives a quick overview of some of the things you can do.

`fsck`, the UNIX file system check program, can be run from a shell script only with the -y or -n options. The manual defines the -y option as follows:

"Assume a yes response to all questions asked by `fsck`; this should be used with extreme caution, as it is a free license to continue, even after severe problems are encountered."

The -n option is safer, but almost uselessly so. This kind of interface is inexcusably bad, yet many programs have the same style. `ftp`, a file transfer program, has an option that disables interactive prompting so that it can be run from a script. But it provides no way to take alternative action should an error occur.

Expect is a tool for controlling interactive programs like these. It solves the `fsck` problem, providing all the interactive functionality noninteractively. Expect is not specifically designed for `fsck` and can handle errors from ftp as well.

The problems with `fsck` and `ftp` illustrate a major limitation in the user interface offered by shells such as sh, csh, and others (which I will generically refer to as "the shell" from now on). The shell does not provide a way of reading output from and writing input to a program. This means a shell script can run `fsck` but only by missing out on some of its useful features. Some programs cannot be run at all from a shell script. For example, `passwd` cannot be run without a user interactively supplying the input. Similar programs that cannot be automated in a shell script are `telnet`, `crypt`, `su`, `rlogin`, and `gdb`. A large number of application programs demand user input.

Expect was designed specifically to interact with interactive programs like these. An Expect programmer can write a script describing the dialogue. Then the Expect program can run the "interactive" program noninteractively. Expect can also be used to automate only parts of a dialogue, since control can be passed from the script to the keyboard and vice versa. This allows a script to do the drudgery and a user to do the fun stuff.

A Very Brief Overview

Expect programs can be written in C or C++, but are almost always written using Tcl (pronounced "tickle," or, if you're uncomfortable using this particular word in a formal setting, "tee cee ell.") Tcl is an interpreted language that is widely used in many other applications. If you already use a Tcl-based application, you will not have to learn a new language for Expect.

Tcl is a very typical-looking shell-like language. There are commands to set variables (set), control flow (if, while, foreach, etc.), and perform the usual math and string operations. Of course, UNIX programs can be called (exec).

Expect is integrated on top of Tcl and provides additional commands for interacting with programs. Expect is named after the specific command which waits for output from a program. The expect command is the heart of the Expect program. The expect command describes a list of patterns to watch for. Each pattern is followed by an action. If the patttern is found, the action is executed.

For example, the following fragment is from a script that involves a login. When executed, the script waits for the strings "welcome", "failed", or "busy" and then it evaluates one of the corresponding actions. The action associated with busy shows how multiple commands can be evaluated. The timeout keyword is a special pattern that matches if no other pattern matches in a certain amount of time.

```
expect {
        "welcome"       break "failed"        abort timeout
        abort "busy" {
                puts "busy" continue } }
```

Total Automation

Earlier I mentioned some programs that cannot be automated with the shell. It is difficult to imagine why you might even want to embed some of these programs in shell scripts. Certainly the original authors of the programs did not conceive of this need. As an example, consider `passwd`.

`passwd` is the command to change a password. The `passwd` program does not take the new password from the command line. Instead, it interactively prompts for it--twice. Here is what it looks like when run by a system administrator. (When run by users, the interaction is slightly more complex because they are prompted for their old passwords as well.)

```
# passwd libes
Changing password for libes on thunder.
New password:
Retype new password:
```

This is fine for a single password. But suppose you have accounts of your own on a number of unrelated computers and you would like them all to have the same password. Or suppose you are a system administrator establishing 1000 accounts at the beginning of each semester. All of a sudden, an automated passwd makes a lot of sense. Here is an Expect script to do just that--automate `passwd` so that it can be called from a shell script.

```
spawn passwd [lindex $argv 0]
set password [lindex $argv 1]
expect "password:"
send "$password\r"
expect "password:"
send "$password\r"
expect eof
```

The first line starts the `passwd` program with the username passed as an argument. The next line saves the password in a variable for convenience. As in shell scripts, variables do not have to be declared in advance. In the third line, the expect command looks for the pattern "password:". expect waits until the pattern is found before continuing.

After receiving the prompt, the next line sends a password to the current process. The \r indicates a carriage-return. (Most of the usual C string conventions are supported.) There are two expect-send sequences because `passwd` asks the password to be typed twice as a spelling verification. There is no point to this in a noninteractive `passwd`, but the script has to do it because `passwd` assumes it is interacting with a human who does not type consistently.

The final command "expect eof" causes the script to wait for the end-of-file in the output of `passwd`. Similar to `timeout`, `eof` is another keyword pattern. This final expect waits for `passwd` to complete execution before returning control to the script.

Take a step back for a moment. Consider that this problem could be solved in a different way. You could edit the source to `passwd` (should you be so lucky as to have it) and modify it so that, given an optional flag, it reads its arguments from the command line just the way that the Expect script does. If you lack the source and have to write `passwd` from scratch, of course, then you will have to worry about how to encrypt passwords, lock and write the password database, etc. In fact, even if you only modify the existing code, you may find it surprisingly complicated code to look at. The `passwd` program does some very tricky things. If you do get it to work, pray that nothing changes when your system is upgraded. If the vendor adds NIS, Kerberos, shadow passwords, a different encryption function, or some other new feature, you will have to revisit the code.

Partial Automation Expect's interact command turns control of a process over to you, so that you can type directly to the process instead of through send commands. Consider `fsck`, the UNIX program I mentioned earlier which checks file system consistency. `fsck` provides almost no way of answering questions in advance. About all you can say is "answer everything yes" or "answer everything no".

The following fragment shows how a script can automatically answer some questions differently than others. The script begins by spawning `fsck`, and then in a loop answering yes to one type of question and no to another. The \\ prevents the next character from being interpreted as a wildcard. In this example, the asterisk is a wildcard but the question mark is not and matches a literal question mark. while 1 { expect { eof {break} "UNREF FILE*CLEAR\\?" {send "y\r"} "BAD INODE*FIX\\?" {send "n\r"} "\\? " {interact +} } } The last question mark is a catchall. If the script sees a question it does not understand, it executes the interact command, which passes control back to you. Your keystrokes go directly to `fsck`. When done, you can exit or return control to the script, here triggered by pressing the plus key. If you return control to the script, automated processing continues where it left off.

Without Expect, `fsck` can be run noninteractively only with very reduced functionality. It is barely programmable and yet it is the most critical of system administration tools. Many other tools have similarly deficient user interfaces. In fact, the large number of these is precisely what inspired the original development of Expect.

The interact command can be used to partially automate any program. Another popular use is for writing scripts that telnet through a number of hosts or front-ends, automatically handling protocols as encountered. When they finally reach a point that you would like to take over, you can do so. For example. you could browse through remote library catalogs this way. Using Expect, scripts can make a number of different library systems seem like they are all connected, rather than different and disconnected.

Intelligently managing interactive programs has been a long-standing problem, traditionally solved by avoidance. Yet the number of interactive programs grows daily and shells have not changed to address this. In contrast, Expect solves these problems directly and with elegance. Expect is a welcome addition to the UNIX workbench.

Don Libes is a computer scientist at the National Institute of Standards and Technology, where he is working on research related to interaction automation. Libes has written more than 75 papers and articles on computer science and is the author of three books: Exploring Expect (O'Reilly & Associates), Life with UNIX (Prentice Hall), and Obfuscated C and Other Mysteries (Wiley).

This article was excerpted from Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.